2 M2: Primitive Datentypen

2.1 Überblick über die primitiven Datentypen in Java[4]

Primitive Datentypen sind in Java unveränderlich eingebunden. Sie besitzen fest definierte Wertebereiche und sind gegen Objekte abgegrenzt.

Die primitiven Datentypen wurden aus Performance-Gründen nicht als Klassen realisiert. Die Bezeichnung primitiv verweist auf die Tatsache, dass primitive Datentypen weder Eigenschaften (Variablen) noch Fähigkeiten (Methoden) besitzen. Es bedeutet somit eher soviel wie einfach aufgebaut. Ihre Anzahl und Datengröße ist eindeutig festgelegt. Neben sechs numerischen Typen existieren der Typ `char` zur Darstellung von Unicode-Zeichen und der Wahrheitswert `boolean`. Einen Überblick gibt die folgende Tabelle:

Typ	Vorzeichen	Größe	Wertebereich
byte	ja	8 bit	-2^7 bis 2^7-1 (-128 bis 127)
short	ja	16 bit	-2^{15} bis $2^{15}-1$ (-32768 bis 32767)
int	ja	32 bit	-2^{31} bis $2^{31}-1$ (-2147483648 bis 2147483647)
long	ja	64 bit	-2^{63} bis $2^{63}-1$ (-9223372036854775808 bis 9223372036854775807)
char	nein	16 bit	16-Bit Unicode Zeichen (0x0000 bis 0xffff (6553510))
float	ja	32 bit V: 1 bit E: 8 bit M: 23 bit	$-3.40282347 \cdot 10^{38}$ bis $3.40282347 \cdot 10^{38}$
double	ja	64 bit V: 1 bit E: 11 bit M: 52 bit	$-1.79769313486231570 \cdot 10^{308}$ bis $1.79769313486231570 \cdot 10^{308}$
boolean	-	8 bit	true/false

Da es gelegentlich nützlich sein kann, einen primitiven Datentyp wie ein Objekt zu behandeln, etwa zum Wandeln eines numerischen Wertes in einen `String`, existieren in Java für jeden primitiven Typ sog. Wrapper-Klassen, die ihn in ein Objekt verpacken und so die Anwendung von Methoden ermöglichen.

`boolean`: Im Gegensatz zu anderen Programmiersprachen ist boolean in Java kein numerischer Typ. Er kann weder inkrementiert/dekrementiert, noch durch numerische Literale repräsentiert werden, sondern ausschließlich durch `true` oder `false`.

[4]Text von https://javabeginners.de/Grundlagen/Datentypen/Primitive_Datentypen.php

2.2 Ganzzahlen

Zunächst schauen wir uns die Codierung von Ganzzahlen in den Typen `byte`, `short`, `int` und `long` an. Die vier Datentypen unterscheiden sich lediglich darin, wie viele Bits für die Speicherung einer Zahl zur Verfügung stehen.

Definition 1. Stellenwertsysteme zur Darstellung von Zahlen:
Zu einer **Basis** b und einer Menge von **Ziffern** $Z = \{0, 1, \ldots b-1\}$ werden Zahlen mit Ziffern $z_i \in Z$ wie folgt dargestellt:

$$\boxed{z_n}\boxed{z_{n-1}}\boxed{z_{n-2}}\cdots\boxed{z_2}\boxed{z_1}\boxed{z_0} = z_n \cdot b^n + z_{n-1} \cdot b^{n-1} + \cdots + z_1 \cdot b^1 + z_0 \cdot b^0$$

Beispiel 1. Die wichtigsten Stellenwertsysteme für die Informatik sind:

- **Dezimalsystem**: $243_{10} = 2 \cdot 10^2 + 4 \cdot 10^1 + 3 \cdot 10^0$
- **Binärsystem**: $1011_2 = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 11_{10}$
- **Hexadezimalsystem**: $\text{A69F}_{16} = 10 \cdot 16^3 + 6 \cdot 16^2 + 9 \cdot 16^1 + 15 \cdot 16^0 = 42\,655_{10} = 1010\ 0110\ 1001\ 1111_2$. Beachte, dass jeweils ein Viererblock im Binärsystem einer Ziffer im Hexadezimalsystem entspricht, da $2^4 = 16$. Deshalb wird das Hexadezimalsystem verwendet, um längere Bitfolgen anschaulicher darzustellen.

Aufgabe 1. Wandle um:

1. vom Dezimal- ins Binär- und Hexadezimalsystem:

   a) 10   b) 100   c) 256   d) 2023

2. vom Binär- ins Dezimal- und Hexadezimalsystem:

   a) 10   c) 0110 0110
   b) 1 0110   d) 11111010101

3. vom Hexadezimal- ins Binär- und Dezimalsystem:

   a) 10   b) D2   c) AFFE

Lösungen:

1. a) $1010_2 = A_{16}$ c) $1\ 0000\ 0000_2 = 100_{16}$
   b) $110\ 0100_2 = 64_{16}$ d) $111\ 1110\ 0111_2 = 7E7_{16}$
2. a) $2_{10} = 2_{16}$ c) $102_{10} = 66_{16}$
   b) $22_{10} = 16_{16}$ d) $2005_{10} = 7D5_{16}$
3. a) $1\ 0000_2 = 16_{10}$ c) $1010\ 1111\ 1111\ 1110_2 = 45\,054_{10}$
   b) $1101\ 0010_2 = 210_{10}$

Definition 2. Ganzzahlige Datentypen werden im **Zweierkomplement (*two's complement*)** dargestellt: Liegt die binäre Codierung einer natürlichen Zahl n in der Anzahl Bits des Datentyps vor, so erhält man die binäre Codierung von $-n$ wie folgt:

1. Invertiere jedes Bit der Codierung von n
2. Addiere 1 dazu (wobei überzählige Bits abgeschnitten werden)

Beispiel 2. -6 als `byte` (=8 Bit):

1. 0000 0110 ist die binäre Codierung von $n = 6$.
2. Invertiere: 1111 1001
3. Addiere 1: 1111 1010 $= -6_{10}$

Aufgabe 2. Rechnen im Zweierkomplement:

1. Codiere als `byte` im Zweierkomplement:

   a) -24 b) -0 c) $-(-6)$

2. Was ist die grösste und kleinste mögliche Zahl, die im Zweierkomplement in 8 Bit codiert werden können?
3. Berechne schriftlich im Binärsystem als `byte`. Behandle dabei Subtraktionen als Addition der Gegenzahl:

   a) $12 + 23$
   b) $8 - 5$
   c) $13 - 105$
   d) $5 \cdot 111$
   e) $66/3$

3 Fliesskommazahlen (float, double)

Aufgabe 1. Betrachte die Klasse `Double.java` und führe sie aus.

1. Führe den Code von Problem 1 aus. Woran scheitert er?
   Kannst du den Code so verändern, dass `0.5` ausgegeben wird? Suche einfache Lösungen! Wenn du eine gefunden hast, probiere aus, ob es noch kürzer geht.

   Das Problem ist, dass `1` und `2` als `int` interpretiert werden. Das kann auf verschiedene Arten geändert werden:
   - Wir können `1` zu einer Fliesskommazahl machen, indem wir `1.0` (oder auch nur `1.`) schreiben.
   - Alternativ funktioniert auch der Zusatz `d` für `double`: `1d`
   - Eine manuelle **Typkonversion** (***type cast***) einer `int`-Zahl `a` kann mit `(double) a` erreicht werden.
   - Wenn eine der Zahlen ein `double` ist, wird die Operation eine `double`-Operation, d.h. die andere Zahl wird vor der Operation automatisch konvertiert. `1. / 2`, `1d / 2` und `(double) 1 / 2` liefern also alle das gewünschte Ergebnis.

2. Entkommentiere den Code von Problem 3 und führe ihn aus. Hast du eine Erklärung für das Verhalten?

   Das Problem ist, dass die Zahlen nicht als Dezimal-, sondern als Binärbruch dargestellt werden. Die nachfolgenden Definitionen und Aufgaben versuchen, Licht ins Dunkel zu bringen.

Definition 3. Fliesskommazahlen werden zur Speicherung als `float` oder `double` zunächst analog zur dezimalen wissenschaftlichen Schreibweise (z.B. $2.71828 \cdot 10^{-23}$) geschrieben - einfach binär statt dezimal:

$$(-1)^S \cdot m \cdot 2^e,$$

wobei S das Vorzeichenbit ist, m die Mantisse (die immer mit einer 1 gefolgt von einem Dezimalpunkt beginnt) und e der Exponent (alles Binärzahlen!).

Davon werden gespeichert:

- S (1 bit)
- $E = e + B$ für einen Biaswert B, so dass E zu einer positiven Ganzzahl wird.
  Für `float` hat E 8 Bits, und $B = 2^{8-1} - 1 = 127$.
  Für `double` hat E 11 Bits, und $B = 2^{11-1} - 1 = 1023$.

Spezialfall: Für den kleinst- und grösstmöglichen Wert von E (also entweder lauter Nullen oder lauter Einsen) gilt das Codierungsschema nicht. Damit werden Werte wie 0, sehr kleine Zahlen, unendlich und `NaN` („*not a number*") codiert.[5]

- M ist m ohne die führende 1 und den Dezimalpunkt (also ebenfalls eine positive Ganzzahl).
  Für `float` hat M 23 Bits, für `double` 52.

Beispiel 3. Der Dezimalbruch 18.4 soll als `float` gespeichert werden.

1. Als Bruch schreiben, kürzen, in Binärbruch umwandeln: $18.4 = \frac{184}{10} = \frac{92}{5} = \frac{101\ 1100_2}{101_2}$
2. Schriftliche Division:

```
 101 1100 : 101 = 1 0010. 0110 0110...
-101
 ----
   0 110
   - 101
   -----
      10 00
      -1 01
      -----
         110
        -101
         ...
```

3. Normalisieren zu $(-1)^0 \cdot 1.\,0010\,0110\,0110\ldots \cdot (2^4)_{10} \overset{\text{binär}}{=} (-1)^0 \cdot 1.\,0010\,0110\,0110\ldots \cdot 10^{100}$
4. $S = 0$
5. Bestimmen von E durch Addieren der Bias zu $e = 4_{10} = 100$:
   $E = e + 127_{10} = e + 111\ 1111 = 1000\ 0011 (= 131_{10})$
6. Ignorieren der Vorkommastelle der Mantisse, 23 Nachkommastellen mathematisch runden: $M = 001\ 0011\ 0011\ 0011\ 0011\ 0011$ (abgerundet, da 0 folgt)
7. Die Bitfolgen für das Vorzeichen $S = 0$, den Exponenten $E = 1000\ 0011$ und die Mantisse $M = 001\ 0011\ 0011\ 0011\ 0011\ 0011$ werden verkettet: 0100 0001 1001 0011 0011 0011 0011 0011.

Aufgabe 2. Wandle die `float`-Zahl 0100 0001 1001 0011 0011 0011 0011 0011 wieder in einen Dezimalbruch um.

Aufgabe 3. Wie viele signifikante *Dezimal*stellen Genauigkeit sind durch `float` bzw. `double` garantiert?

[5] Mehr Informationen siehe https://de.wikipedia.org/wiki/IEEE_754.

Mit der führenden 1 haben wir $M+1$ signifikante Binärstellen, d.h. 24 für `float` und 53 für `double`.
`float`: $\log_{10}(2^{24}) \approx 7.225 \Rightarrow$ 7-8 signifikante Dezimalstellen.
`double`: $\log_{10}(2^{53}) \approx 15.955 \Rightarrow$ 15-16 signifikante Dezimalstellen.

Aufgabe 4. Bestimme die grösste und die kleinste positive Zahl, die als `float` bzw. `double` nach dem Standardschema gespeichert werden kann

Lösung: Für positive Zahlen gilt $S=0$, was aber für die Fragestellung irrelevant ist. E kann nicht nur aus Nullen oder nur aus Einsen bestehen, da diese „Exponenten“ ja für Spezialcodierungen reserviert sind. Deshalb werden diese Werte jeweils um 1 erhöht bzw. vermindert.

- Für die kleinste Zahl ist $E=1$ und $M=0$, also $m=1$:
  - In `float` hat die Zahl $e=-126$, ist also gleich $1\cdot 2^{-126} \approx 1.18\cdot 10^{-38}$.
  - In `double` hat die Zahl $e=-1022$, ist also gleich $1\cdot 2^{-1022} \approx 2.23\cdot 10^{-308}$.
- Für die grösste Zahl gilt:
  - In `float` ist $E = 1111\ 1110 = 254_{10}$, also $e = 254-127 = 127$,
    $M = 2^{23}-1$, also $m = \frac{2^{24}-1}{2^{23}} = 2-2^{-23}$
    und somit die grösste Zahl $(2-2^{-23})\cdot 2^{127} \approx 3.40\cdot 10^{38}$.
  - In `double` ist $E = 111\ 1111\ 1110 = 2046_{10}$, also $e = 2046-1023 = 1023$
    $M = 2^{52}-1$, also $m = \frac{2^{53}-1}{2^{52}} = 2-2^{-52}$
    und somit die grösste Zahl $(2-2^{-52})\cdot 2^{1023} \approx 1.80\cdot 10^{308}$.

Aufgabe 5. Versuche möglichst genau zu erklären, was bei der `double`-Operation `0.1 + 0.2` geschieht!

Hinweise:

0. 0001 1001 1001 1001 1001 1001 1001 1001 1001 1001 1001 1001 1001 1010

0. 0011 0011 0011 0011 0011 0011 0011 0011 0011 0011 0011 0011 0011 010

0. 0100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1101 00

0. 0100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 11